

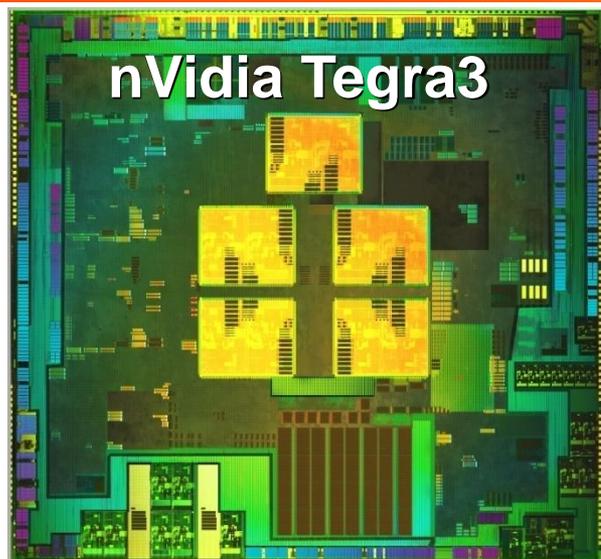
Application analysis for parallelization on multi-core devices

Jos van Eijndhoven
jos@vectorfabrics.com

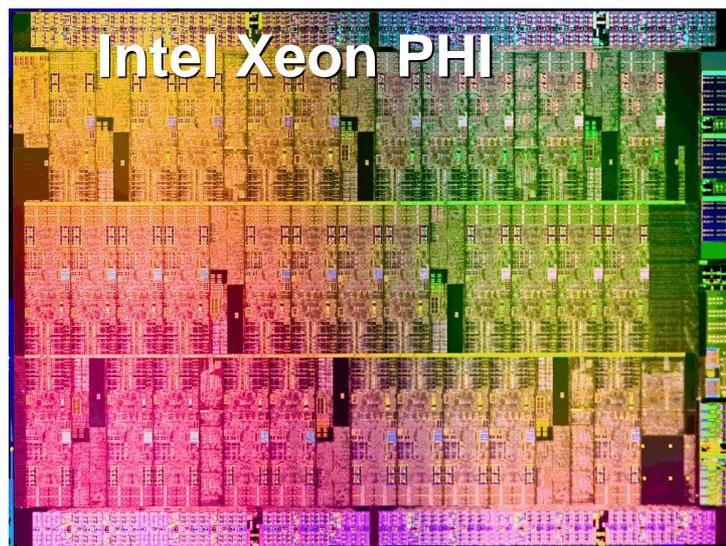
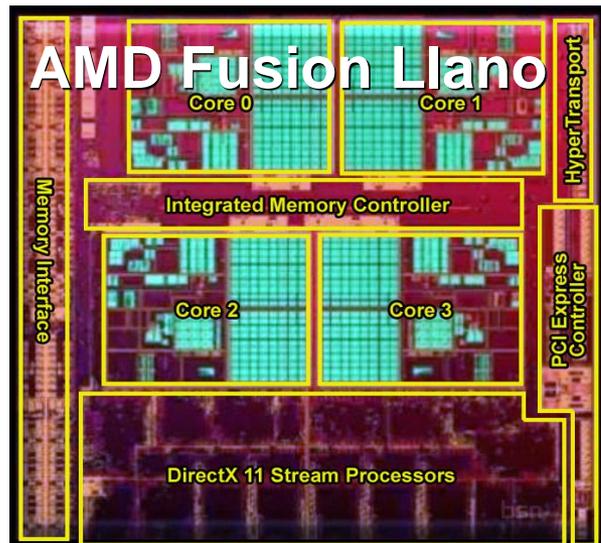
HiPEAC Computing Systems, Ghent, Belgium
Oct 16, 2012



Multi-core processors are here to stay



- To make use of growing transistor count
- To allow run-time trade-offs between performance and power



Multi-core in Mobile

- 2 cores:
Assume the OS provides **multiple processes** and/or kernel threads for workload
- 4 cores (and beyond):
Requires **multi-threaded applications**
 - To obtain sufficient concurrent workload
 - To obtain top user experience

Who makes such applications??

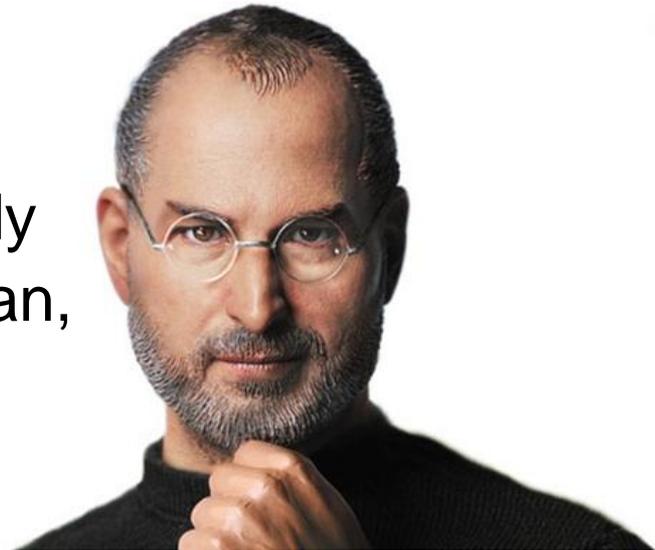
Creating parallel programs is hard...

Herb Sutter, chair of the ISO C++ standards committee,
Microsoft:

“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren’t possible, and discovers that they didn’t actually understand it yet after all”

Steve Jobs, Apple:

“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two yeah; four not really; eight, forget it.”

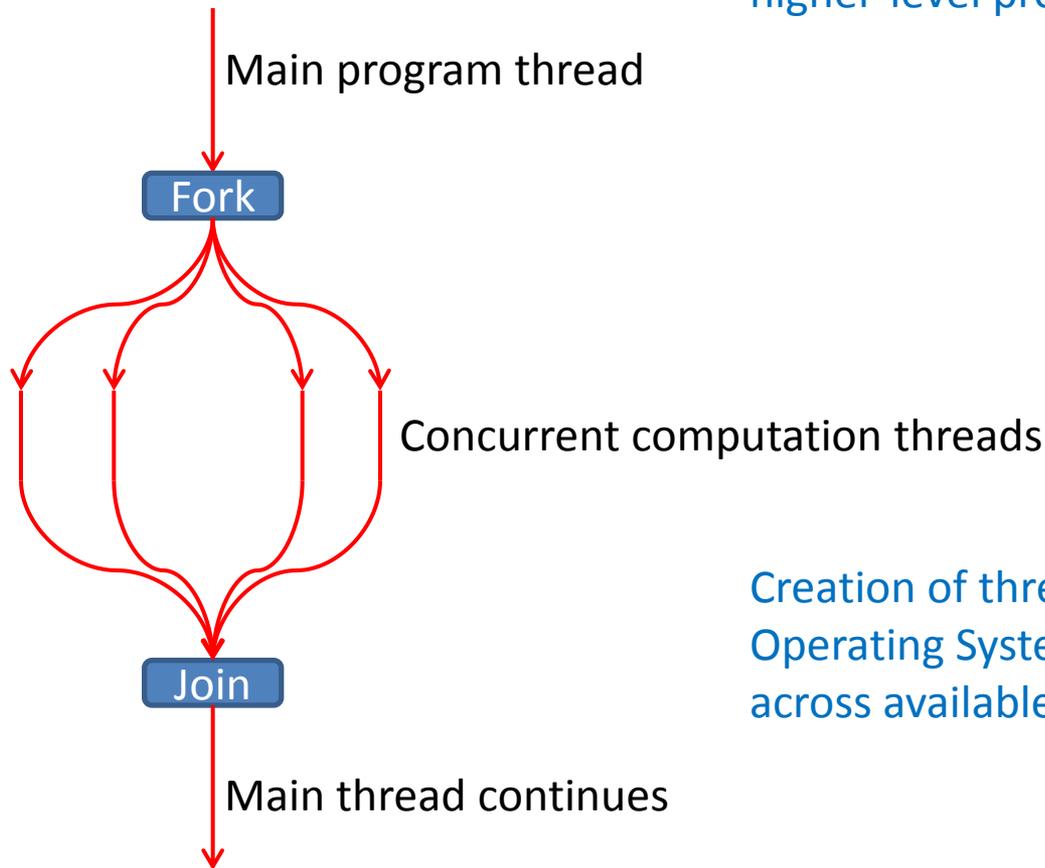


Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
 - Data-parallelization with reduction expressions
 - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Creating multi-threaded concurrency

Basic fork-join pattern, created through different higher-level programming constructs



Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors!

Parallelization – two partitioning options

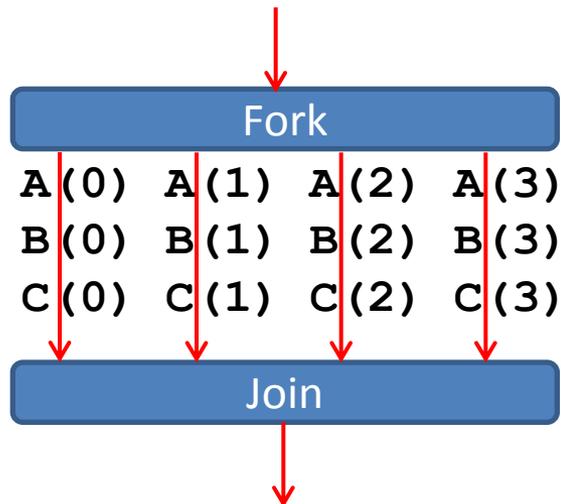
Source code:

```
for (i=0; i<4; i++) {  
    A(i);  
    B(i);  
    C(i);  
}
```

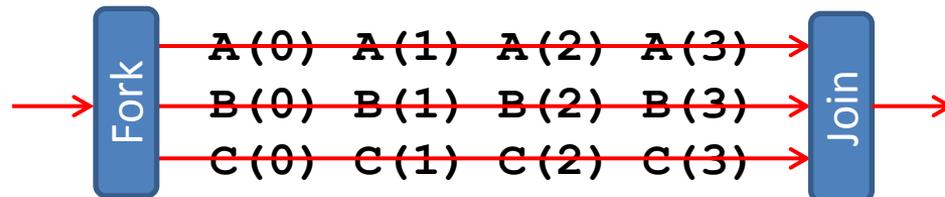
Sequential execution order:

A(0) A(1) A(2) A(3)
B(0) B(1) B(2) B(3)
C(0) C(1) C(2) C(3)

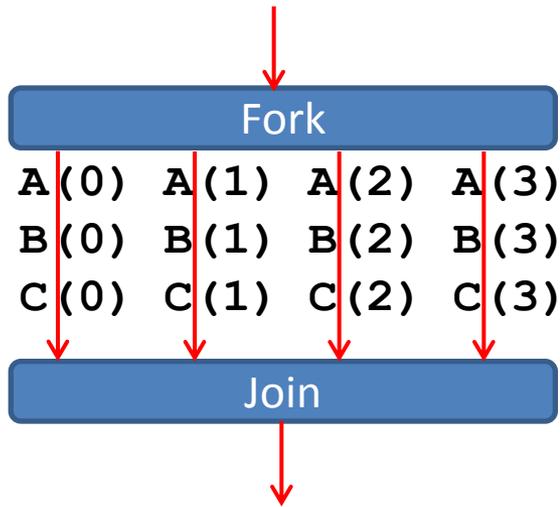
Data partitioning:



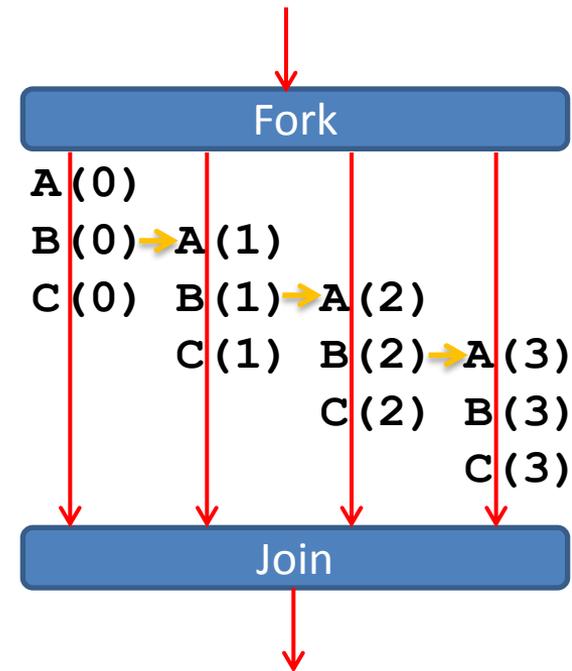
Task partitioning:



Issue: Data dependencies



Maybe, $B(i)$
produces a value
that is used by
 $A(i+1) \dots$



Adjust program source for parallelization:

- When feasible, remove inter-thread data dependencies
- Implement required data synchronization

Consciously choose task versus data partitioning, check dependency analysis!

Category 1: Data dependencies

Variable assigned in loop body, used in later iteration

```
// search linked-list for matching items
// save matches in 'found' array of pointers
for (p = head, n_found = 0; p; p = p->next)
    if (match_criterion(p))
        found[n_found++] = p;
```

Cannot (easily/trivially) spawn data-parallel tasks!

- No direct parallel access to list members ***p**
- No direct way to assign index to matched item **n_found**
- Maybe more problems hidden in **match_criterion**

Category 2: Anti dependencies

Storage location used in loop body, shared over iterations

```
// convert table with floats to strings
char word[64];
for (i=0; i<N; i++)
{
    sprintf( word, "%g", table_float[i]);
    table_string[i] = strdup( word);
}
```

- Anti-dependencies are resolved by duplicating storage locations (thread-local storage)
- Need to make multiple copies of `word[]` space

Category 3: Control dependencies

Control flow can give order constraints that hinders parallelization:

```
// No creation of work beyond some point
for (i=0; i<N; i++)
{
    if (special_condition(i))
        break;
    table[i] = workload(i);
}
```

Since multiple threads proceed at non-determined mutual speed, above test risks violation in a data-parallel loop.

Note: C++ exceptions certainly belong to this category

Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
 - Data-parallelization with reduction expressions
 - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Can do: reduction data dependencies

- Reduction expressions: accumulate results of loop bodies with commutative operations
- Freedom of re-ordering allows to break sequential constraints

```
// conditionally accumulate results
```

```
int acc = 0;  
for (i=0; i<N; i++)  
{  
    int result = some_work(i);  
    if (some_condition(i))  
        acc += result;  
}
```

```
...use of acc ...
```

- Commutative operations are basic math like +, *, &&, &, ||, but also more complex operations like 'add to set'.
- Three(?) different methods to handle these ...

Three methods for reduction dependencies

- Create thread-local copies of the accumulator. Accumulate over local copy in each thread. Merge the partial accumulators after thread-join.

Eg. created automatically by:

```
#pragma omp parallel for reduction(...)
```

- Maintain single accumulator, synchronize updates through atomic operations. Eg. in C++11:

```
atomic_add_fetch( &acc, result);
```

- Maintain single accumulator, synchronize updates through protection by acquiring and releasing semaphores.

Eg. Used by C++ Intel TBB:

```
concurrent_unordered_set<..> s;  
s.insert(...);
```

Example data partitioning

```
int sum = 0;
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Distribute the workload over multiple cores.
- Each core handles part of the loop index space.

```
int sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Workload scales nicely across multiple cores
- Easy to write down 😊, but hard to grasp all consequences!
- Dangerous, might cause extremely hard-to-track bugs! 😞

PAREON: Parallelization Analysis

The screenshot displays the PAREON software interface, which is used for parallelization analysis. The interface is divided into several panels:

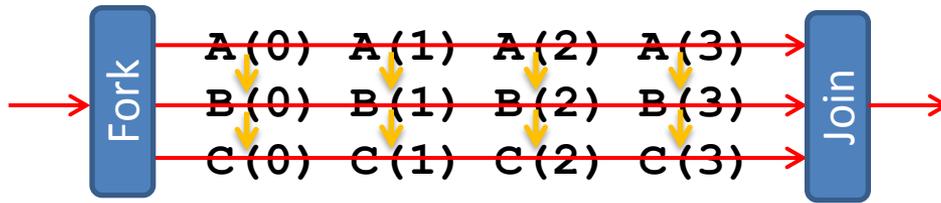
- Partitioning candidates - Loop_38:** This panel shows CPU data partitioning for the 'Loop_38' task. The number of threads is set to 4. The global speedup is 2.3, and the global overhead is 6%. The table below summarizes the partitioning candidates.
- Properties / My changes:** This panel shows the properties of the 'Loop_38' task, including iteration count, iteration time, and iteration statistics.
- 2D-Profile / Schedule:** This panel shows a 2D profile of the execution. The top part shows a 'Schedule overview' with a bar representing 'Execution - 99 %'. The bottom part shows a 'Schedule execution (prologue - steady state - epilogue)' with a grid of iterations. A blue callout box points to the grid with the text: "Note: this is a *preview* on a potential parallelization".

Partitioning candidates	Speedup	Overhead	Streams
Loop_38	3.9	1 %	1

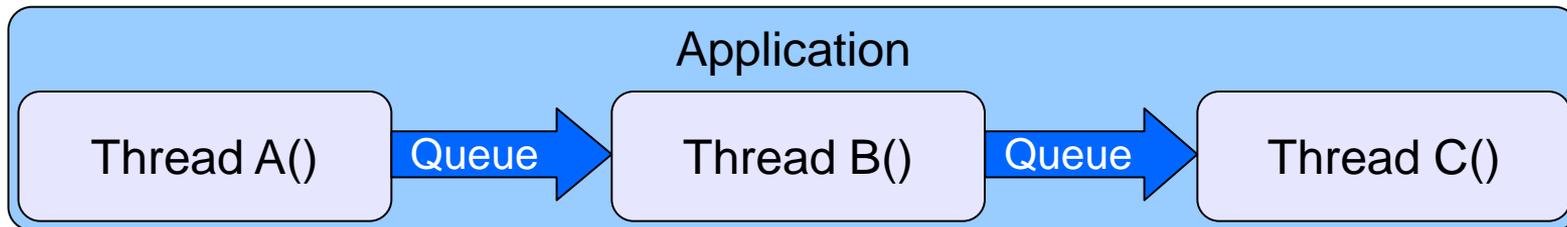
Property	Value
Loop_38 (sgtf2_)	
Loop	Loop_38 (sgtf2_)
Iteration count	150
Iteration time	
Iteration statistics	
Computation time	85.3 us (92.7 %)
Memory penalty	6.8 us (7.3 %)
Load count	15770
Store count	7802
Instruction count	104658
Mapped to Instance	ARM-A9
Source location	sgtf2.c:141-185
Line coverage	79.2 %
Uncovered lines	

Pipelining: Data deps & functional partitioning

Functional partitioning with inter-thread dependencies:



Producer-Consumer pattern:



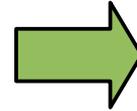
Queue implementation solves dependencies:

- **Synchronize Data dependencies:** Consumer thread waits for available data (stalls until queue is non-empty)
- **Solve Anti dependencies:** Producer thread creates next item in next memory location (prevents overwriting previous value)

Example functional partitioning

```
int A[N][M];
```

```
while (...)
{ produce_img();
  consume_img();
}
```



```
produce_img()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}
```

```
consume_img()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}
```

```
Thread1:
```

```
while (...)
  produce_img();
```

```
Thread2:
```

```
while (...)
  consume_img();
```

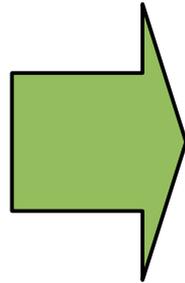
Function pipelining: synchronization

```
int A[N][M];

while (...)
{ produce_img();
  consume_img();
}

produce_img()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}

consume_img()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}
```



```
Thread1: ...
```

```
Thread2:...
```

```
concurrent_queue<int> qA;
```

```
produce_img()
{ for (i ...)
  for (j ...)
    qA.push(...);
}
```

```
consume_img()
{ for (i ...)
  for (j ...)
    qA.pop(&...);
}
```

Conversion to queues becomes more difficult when data items are not always assigned and referenced exactly once in order!

PAREON: Pipeline dependency analysis

The screenshot displays the PAREON web interface for pipeline dependency analysis. The browser address bar shows the URL `https://jos.vectorfabrics.com/28000/html/Main.html`. The interface includes a top navigation bar with tabs for Project, Profile, Partitions, My changes, Log, 2D-Profile, Schedule, Architecture, and PGtrain.c. A sidebar on the left contains a 'Data partitioning candidates' section with a warning for 'Loop_36507' and a 'Functional partitioning - Loop_36507' table. The main area shows a 3D visualization of pipeline stages with a '1.0 SPEED-UP' indicator. A bottom section displays a dependency graph with a legend for Compute, Memory, Streaming, and Anti-dependencies. A callout bubble points to the graph with the text 'Potential pipelining showed in colors, with resulting Fifo's'. A right sidebar contains 'Cheat sheets' and an 'Introduction' section.

Project Profile Partitions My changes Log 2D-Profile Schedule Architecture PGtrain.c Labs View Help Close Cheat sheets

https://jos.vectorfabrics.com/28000/html/Main.html

Data partitioning candidates

- Loop_36507 cannot be subjected to data partitioning: There are 1 loop-carried memory clusters that you have chosen not to ignore: (memory_cluster_25).

Functional partitioning - Loop_36507

Partition id	Threads	Speedup	Streams	Apply
Partition 1	4	2.6	3	Apply
Partition 2	4	2.6	3	Apply
Partition 3	3	2.4	2	Apply
Partition 4	3	2.2	2	Apply
Partition 5	2	1.2	0	Apply

Properties

Function: `pgmain_PG_SELECT_3_p_full_cal`

Line coverage: 30.8 %

Uncovered lines

Invocation time

- Estimated: 38.188 ns
- Constraint: <click to edit>

Invocation statistics

- Computation time: 28.812 ns (75.5 %)
- Memory penalty: 9.375 ns (24.5 %)
 - DRAM traffic: 0 B
 - DRAM bandwidth: 0 B
- Data cache statistics
 - Penalties
 - Level 1: 9.375 ns
 - Level 2: 0 ps

Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
 - Data-parallelization with reduction expressions
 - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Concurrent C/C++ programming: Pitfalls

Risc introduction of functional errors:

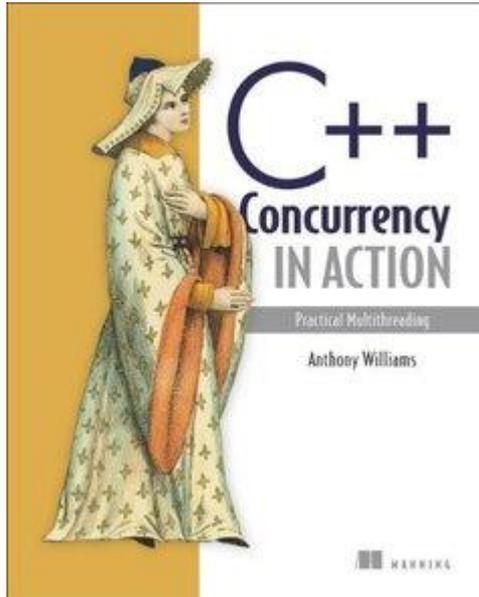
- Overlooking use of shared/global variables (deep down inside called functions, or inside 3rd party library)
- Overlooking exceptions that are raised and caught outside studied scope
- Incorrect use of semaphores: flawed protection, deadlocks

Unexpected performance issues:

- Underestimation of time spent in added multi-threading or synchronization code and libraries
- Underestimation of other penalties in OS and HW (inter-core cache penalties, context switches, clock-frequency reductions)

Parallel programming remains hard!

Concurrent programming remains hard



- C++11 standardizes valuable primitives
- Provides good insight in C++ concurrency
- Warns for many subtle problems
- From a research point-of-view, shows that C++ is not a nice language to design concurrency.

Development of parallel code

Guidelines:

- Base upon a sequential program:
functional and performance reference
- Apply higher-level parallelization patterns and primitives:
clear semantics, re-use code, reduce risk
- Use tooling for analysis and verification
 - Prevent introduction of hard-to-find bugs
 - Prevent recoding effort that does not perform

Managable development process!

PAREON step 1: Code instrumentation

Build application with compiler that inserts instrumentation:

- Creates instrumentation for run-time tracing of application activity (function entry/exit, loop entry/exit, ld/st addresses)
- To support run-time data-dependency analysis
- Also support code coverage analysis

PAREON 2: run-time dependency analysis

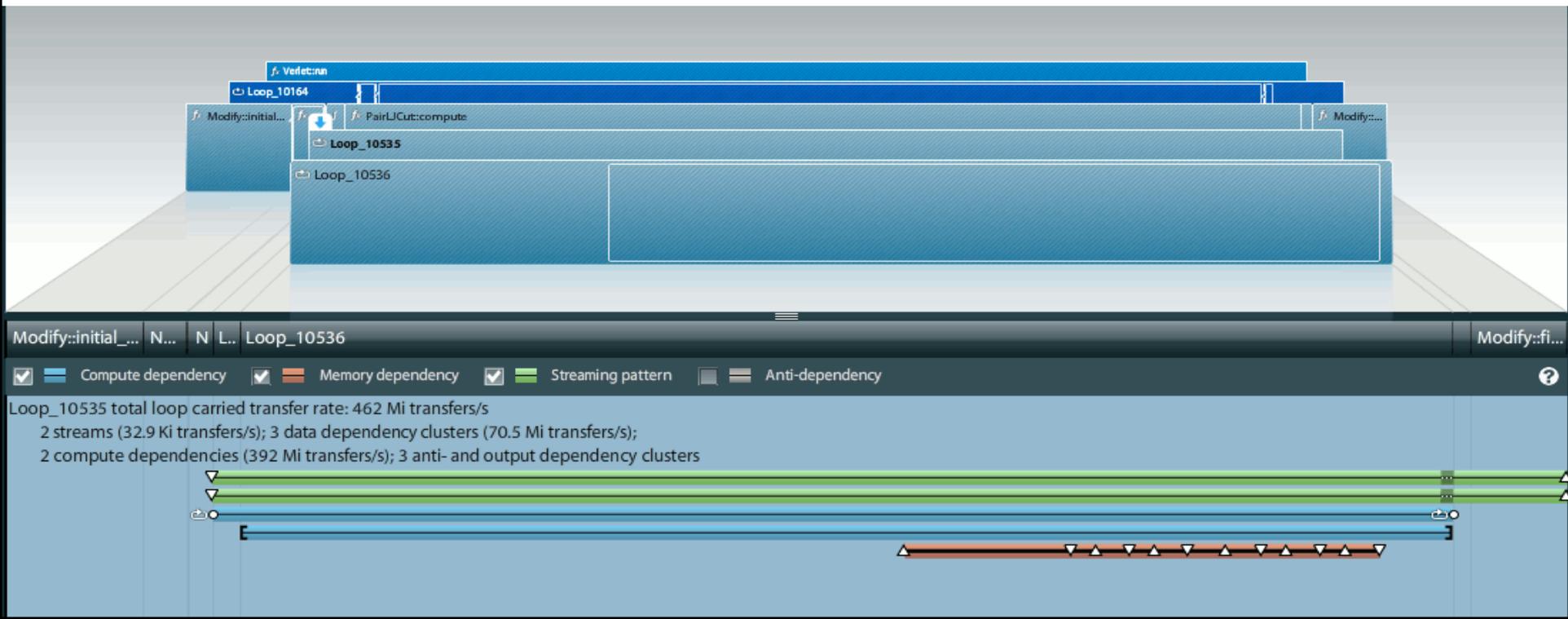
Execute instrumented program with test input data:

- Trace analysis detects dependencies between loads & stores at different program locations to same memory address.
- Differentiate loop-inbound, loop-carried and loop-outbound dependencies
- Relate with stack grow/shrink and heap malloc/free to break non-functional address re-use.
- Handle all scalar register-mapped data dependencies by static code analysis.

PAREON 3: find concurrency opportunities

GUI to browse loops with high workload and parallelization opportunities:

- Provide workload estimate and reachable speed-up
- Match detected dependencies with higher-level parallelization patterns for resolving (...)
- Prevent loop parallelization with unresolved dependencies

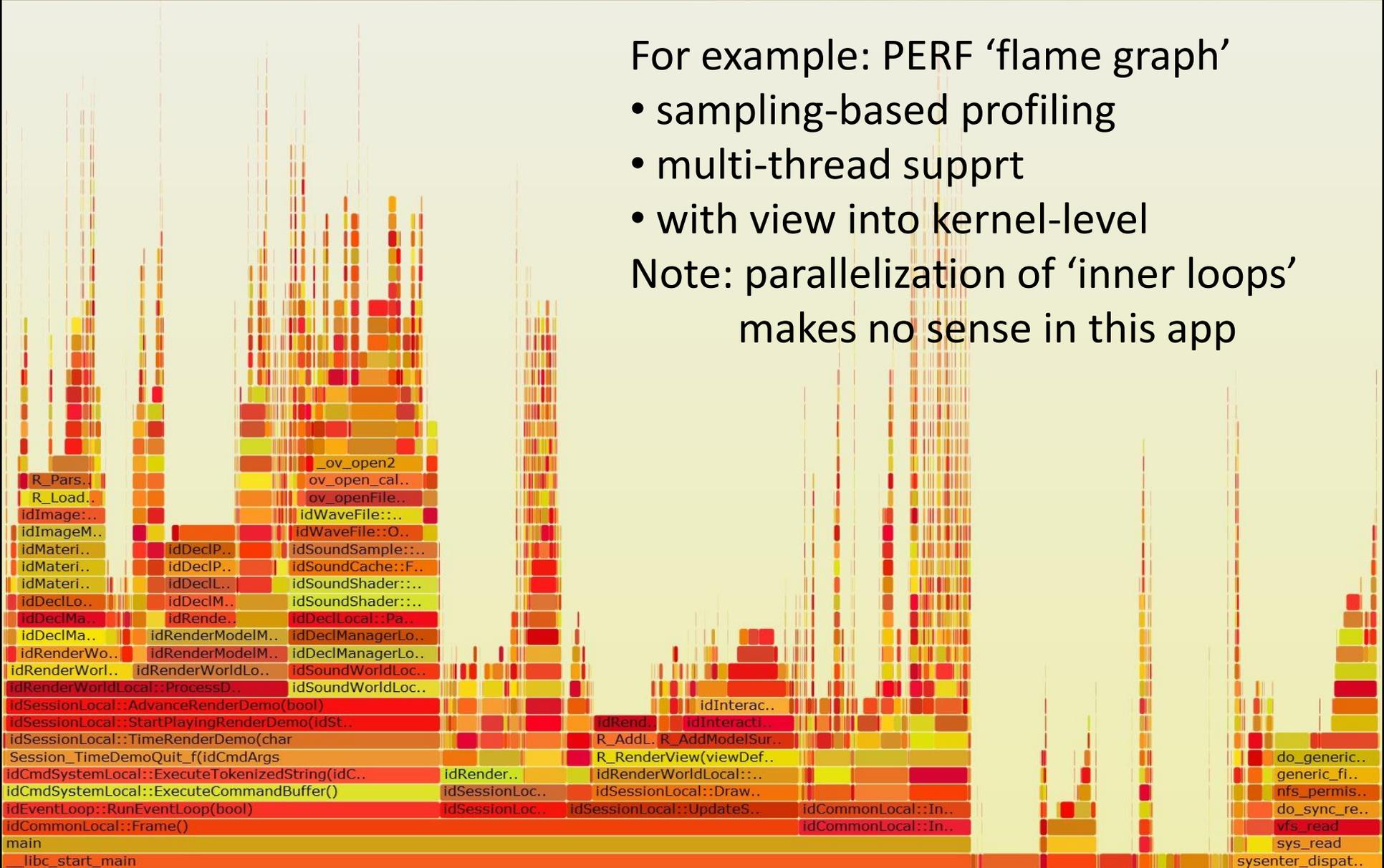


Performance Verification

For example: PERF 'flame graph'

- sampling-based profiling
- multi-thread support
- with view into kernel-level

Note: parallelization of 'inner loops' makes no sense in this app

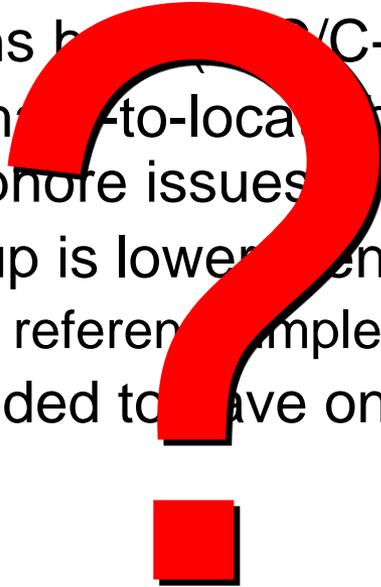


Conclusion

- **Today's gap:** multi-core CPUs are everywhere, yet multi-threaded programming remains hard (in C/C++):
 - Risk of creating hard-to-locate bugs regarding dynamic data races and semaphore issues
 - Obtained speedup is lower than expected
- A sequential functional reference implementation helps to set a baseline
- Proper tooling is needed to save on edit-verify development cycles

Conclusion

- **Today's gap:** multi-core CPUs are everywhere, yet multi-threaded programming remains hard (C/C++):
 - Risk of creating hard-to-localize bugs regarding dynamic data races and semaphore issues
 - Obtained speedup is lower than expected
- A sequential functional reference implementation helps to set a baseline
- Proper tooling is needed to move on edit-verify development cycles



Thank you

Check www.vectorfabrics.com for a free demo on concurrency analysis

